Logical Methods in Computer Science
Vol. 12(2:6)2016, pp. 1–28
www.lmcs-online.org

Submitted Jul. 3, 2014
Published Jun. 13, 2016

CERTIFIED CONTEXT-FREE PARSING: A FORMALISATION OF VALIANT'S ALGORITHM IN AGDA

JEAN-PHILIPPE BERNARDY AND PATRIK JANSSON

Chalmers University of Technology & University of Gothenburg, Sweden
e-mail address: {bernardy, patrikj}@chalmers.se

Abstract. Valiant (1975) has developed an algorithm for recognition of context free languages. As of today, it remains the algorithm with the best asymptotic complexity for this purpose. In this paper, we present an algebraic specification, implementation, and proof of correctness of a generalisation of Valiant's algorithm. The generalisation can be used for recognition, parsing or generic calculation of the transitive closure of upper triangular matrices. The proof is certified by the Agda proof assistant. The certification is representative of state-of-the-art methods for specification and proofs in proof assistants based on type-theory. As such, this paper can be read as a tutorial for the Agda system.

Introduction

Context-free grammars [Chomsky, 1957] are the standard formalism to express and study the syntactic structure of programming languages. While numerous algorithms are used for parsing context-free inputs, the subject of this paper is a generalisation of the recognition algorithm discovered by Valiant [1975]. (For simplicity we call the generalisation simply "Valiant's algorithm" below.)

Valiant's algorithm has many qualities. First, it is efficient: it is the parsing algorithm with the best worst-case asymptotic complexity [Lee, 2002] ($O(n^3)$). Further, it has recently been identified that its performance in common average cases is also excellent [Bernardy and Claessen, 2015]: in the presence of a hierarchical input, and given some care in the representation of grammars and data structures, it behaves linearly, and can be parallelized. Second, Valiant's algorithm is abstract and general. While its main application is parsing, it solves the problem of finding the transitive closure of a generalised relation W given as an upper-triangular matrix, where the underlying element operation is not necessarily associative. Third, Valiant's algorithm is relatively simple: we will see that it can be expressed as two mutually defined functions by induction on the size of the matrix.

The combined qualities of Valiant's algorithm (importance of the problem solved, efficiency, generality and simplicity) make it, in our opinion, one of the top ten algorithms that every computer scientist should learn. As such, it is a good candidate for being given a

2012 ACM CCS: [**Theory of computation**]: Logic—Automated reasoning; Formalisms—Rewrite systems.

Key words and phrases: Context-Free Parsing, Valiant's algorithm, Proof, Agda.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-12(2:6)2016

fully certified-correct implementation. Such an implementation is the main contribution of this paper.

The medium chosen for our proof is the Agda proof assistant [Norell, 2007]. One motivation for this choice is that Agda is based on type-theory, and using type-theory as a core means that our proof can be verified by a type-checker, which can itself be subject to formal verification. This chain of certification, relying ultimately on a small trusted base shared with many other proofs makes us very confident that our development is correct. (Even though the core of Agda is not currently verified, there is ongoing effort in this direction.)

A secondary goal of this work is to provide an exemplary proof: we have taken particular care to make the proof approachable. In particular: 1. the algorithm is derived from its specification (instead of being first implemented and proved after the fact). 2. the core of formal proof is close to a semi-formal proof developed earlier [Bernardy and Claessen, 2013]. Further, because we assume little knowledge of the Agda proof assistant, our development can be used as a tutorial on algorithm specification and derivation in Agda.

The rest of the paper is organised as follows. In section 1, we review how Valiant's algorithm reduces parsing to the computation of transitive closure. Section 2 introduces Agda's syntax and some basic formalisation concepts. Sections 3 to 8 contain the formal development. We review related work in section 9 and conclude with possible extensions in section 10.

1. Parsing as Transitive Closure

In this section we review the basic idea underlying Valiant's algorithm, namely that context-free parsing can be specified as computing a transitive closure.

1.1. **Chomsky Normal Form.**

The simplest implementation of Valiant's algorithm takes as input a grammar in Chomsky Normal Form [Chomsky, 1959]. In Chomsky Normal From, hereafter abbreviated CNF, the production rules are restricted to one the following forms.

$$A_0 ::= A_1 A_2 \qquad \text{(binary)}$$
$$A ::= t \qquad \text{(unary)}$$
$$S ::= \epsilon \qquad \text{(nullary)}$$

Any context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$ generating a given language can be converted to a grammar $\mathcal{G}'$ in CNF generating the same language. (N is the set of non-terminals, Σ is the alphabet, P is the set of productions, S is the start symbol and we use A to range over elements of N.) Hence we will assume from now on a grammar provided in CNF. Moreover, because it is easy to handle the empty string specially, we conventionally exclude it from the input language and thus exclude the nullary rule $S ::= \epsilon$ from the set of production rules. The reader avid of details is directed to Lange and Leiß [2009] for a pedagogical account of the process of reduction to CNF.

Given a grammar specified as above, the problem of parsing is reduced to finding a binary tree such that each leaf corresponds to a symbol of the input (and a suitable unary rule) and such that each branch corresponds to a suitable binary rule. Essentially, parsing is equivalent to considering all possible bracketings of the input, and finding one (or more) that form a valid parse.

1.2. **Charts.** Let w be a vector of input symbols. We define the operations $0, +, \cdot$ and σ as follows.

Definition 1.1 $(0, +, \cdot$ on $\mathcal{P}(N))$**.**

$$0 = \varnothing \qquad x + y = x \cup y \qquad x \cdot y = \{A_0 \mid A_1 \in x, A_2 \in y, (A_0 ::= A_1 A_2) \in P\}$$
$$\sigma_i = \{A \mid (A ::= w[i]) \in P\}$$

The above operations can be lifted to matrices, in the usual way (and we do so formally in section 6.) The $(\cdot)$ operation fully characterises the binary production rules of the grammar, while σ captures the unary ones. We will then use a matrix of sets of non-terminals to record which non-terminals can generate a given substring. Such a matrix is called a chart. If C is a (complete) chart, $A \in C_{ij}$ iff $A \xrightarrow{*} w[i..j]$. See figure 1 for an illustration. The operation σ is used to construct an initial chart $I(w)$ such that

$$I(w)_{i,i+1} = \sigma_i$$
$$I(w)_{i,j} = 0 \qquad \text{if } j \neq i + 1$$

The matrix $W = I(w)$ is a partial chart: it contains the correct non-terminals for strings of length one, stored at positions $(i, i + 1)$ in the chart. All other positions are empty (zero). Computing the transitive closure of W will complete the chart; so that at position $(0, n)$ one will find the non-terminals generating the whole input.

Definition 1.2 (Transitive closure)**.** If it exists, the transitive closure of a matrix W, written W^+, is the smallest matrix C such that

$$C = W + C \cdot C$$

The equation means that C contains all possible associations of W multiplied by itself [1]:

$$\begin{aligned} C &= W + C \cdot C \\ &= W + (W + C \cdot C) \cdot (W + C \cdot C) \\ &= W + WW + W \cdot (C \cdot C) + (C \cdot C) \cdot W + (C \cdot C) \cdot (C \cdot C) && \text{by distributivity} \\ &= \dots \\ &= W + WW + W \cdot (W \cdot W) + (W \cdot W) \cdot W + (W \cdot W) \cdot (W \cdot W) + \dots && \text{by dist. \& comm.} \end{aligned}$$

Therefore, if a parse tree (possible bracketing) exists, the algorithm will find it. Furthermore, because C is the least matrix satisfying the equation, it will not contain any non-terminal which does not generate the input.

The above procedure specifies a recogniser: by finding the closure of $I(w)$ one finds if w is parsable, but not the corresponding parse tree. However, one can obtain a proper parser by using sets of parse trees (instead of non-terminals) and extending $(\cdot)$ to combine parse trees.

[1] If $(\cdot)$ were associative, then a simpler formula for the transitive closure could be given, and a much more efficient technique could be used to compute it, but then all bracketings would be equivalent and $(\cdot)$ could not capture the binary rules of a context-free grammar.

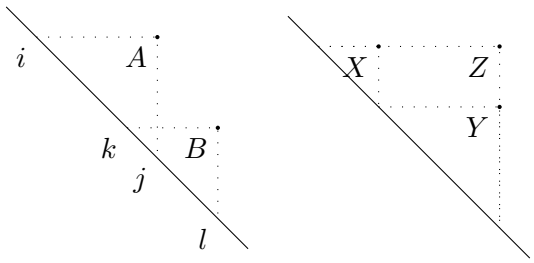

Figure 1: Example charts. In each chart a point at position (r, c) corresponds to a substring starting at r and ending at c. The first parameter (r for row) grows downwards and the second one (c for column) rightwards. The input string w is represented by the diagonal line. Dots in the upper-right part represent non-terminals. The first chart witnesses $A \xrightarrow{*} w[i..j]$ and $B \xrightarrow{*} w[k..l]$. An instance of the rule $Z ::= XY$ is exemplified on the second chart.

2. Agda preliminaries

This section introduces the elements of Agda necessary to understand the upcoming sections. We present both the language itself, and some definitions which are part of the standard library. Throughout the paper, we will use a literate-programming style. The body of this section is a single module Preliminaries which contains "specification building blocks" — a number of definition for later use.

```
module Preliminaries where
```

In Agda code, scoping is indicated by indentation. Indentation is rather hard to visualise in a paper which spans over several pages, so we will instead give scoping hints in the text when necessary. The scope of the current module extends to the end of the section.

Propositions-as-types. The philosophy behind Agda is that each proposition is expressed as a type. That is, proving that a proposition P holds means finding an inhabitant (an element) of P (read as a type). With this in mind, we define the type of relations over an underlying type A as functions mapping two elements of type A to a another type (a Set in Agda parlance).

```
Rel : Set → Set₁
Rel A = A → A → Set
```

Hence, an element of type Rel A is a binary relation on A. For example Rel A will be inhabited by equivalence relations and orderings. A consequence of the above definition is that our relations are constructive: to show that a pair (x, y) is in a relation R we must provide a witness of type R x y. (In passing, the type of relations is a so-called big set (Set_1), because it contains Sets itself. This distinction is necessary for the consistency of the logical system.)

As another example, we can define the existential quantifier connective as follows:

```
record ∃ {A : Set} (P : A → Set) : Set₁ where
  constructor _,_
  field
    proj₁ : A
    proj₂ : P proj₁
```

That is, the existence of an element satisfying P can be written ∃ (\x → P x) (or, equivalently, ∃ P), and proving this proposition means to find a witness x : A and an inhabitant p : P x. The constructor keyword here introduces the name _,_ as the two-argument constructor of the record type. Infix operators are declared using underscores on both sides of the name so an infix comma (,) can now be used as in the following example: (x , p) : ∃ P. Note that only the last argument (P) of the ∃ symbol is written, the first (A) is left for Agda to infer: we say that it is *implicit*. Implicit parameters are marked by placing them in braces at the declaration site. An implicit argument can be supplied explicitly at a call site, if the programmer encloses it with braces. This syntax can be useful if Agda fails to infer the argument in a certain context.

Entire relations. A binary relation from a set A to a set B is called *entire* if every element of A is related to at least one element of B, and we can encode this definition as follows.

```
Entire : {A B : Set} → (_R_ : A → B → Set) → Set
Entire _R_ = ∀ a → ∃ λ b → a R b
```

Here, again, the use of underscores around R makes it an infix operator (in its scope). Fixity is just a presentation issue, so an equivalent, but shorter, definition is Entire R = ∀ a → ∃ (R a) where R is prefix and R a is a partial application. A consequence of proving that a relation R is entire in our constructive setting is that we get a function contained in R. We can extract the function using the first field of the ∃ record.

```
fun : {A B : Set} → {_R_ : A → B → Set} → Entire _R_ → A → B
fun ent a = proj₁ (ent a)
```

The proof that the function is contained in R can be obtained from the second field of ∃:

```
correct : {A B : Set} → {_R_ : A → B → Set} → (ent : Entire _R_) →
          let f = fun ent in ∀ {a : A} → a R (f a)
correct ent {a} = proj₂ (ent a)
```

The above pattern generalises to relations of any number of arguments. In this paper we need the following version:

```
Entire3 : {A B C D : Set} → (R : A → B → C → D → Set) → Set
Entire3 R = ∀ x y z → ∃ (R x y z)
```

with corresponding definitions of fun3 and correct3.

Uniqueness. An element of UniqueSolution _≃_ P is a proof that the predicate P has (at most) a *unique* solution relative to some underlying relation _≃_.

```
UniqueSolution : {A : Set} → Rel A → (A → Set) → Set
UniqueSolution _≃_ P = ∀ {x y} → P x → P y → x ≃ y
```

A proof usP : UniqueSolution _≃_ P is thus a function which given two hidden arguments of type A and two proofs that they satisfy P returns a proof that they are related by _≃_.

Least solutions. In optimisation problems, one often wants to find the least solution with respect to some order _≤_. We use LowerBound _≤_ P for the predicate that holds for an a : A iff a is smaller than all elements satisfying P. If a lower bound is in the set (satisfies the predicate P) it is called *least*.

```
LowerBound : {A : Set} → Rel A → (A → Set) → (A → Set)
LowerBound _≤_ P a = ∀ z → (P z → a ≤ z)

Least : {A : Set} → Rel A → (A → Set) → (A → Set)
Least _≤_ P a = P a × LowerBound _≤_ P a
```

Note that a proof alP : Least _≤_ P a is a pair of a proof that a is in P and a function albP : LowerBound _≤_ P a. And, in turn, albP is a function that takes any z : A (with a proof that z is in P) to a proof that a ≤ z.

Records and modules. The upcoming proof makes extensive use of *records*, which we review now in detail. Agda record types contain fields and helper definitions. Fields refer to data which is stored in the record, while helper definitions provide values which can be computed from such data. Because Agda treats proofs (of propositions) as data, one can require the fields to satisfy some laws, just by adding (proofs of) those laws as fields. Our first record type example, ∃ P, has two fields; one element proj_1 and a proof that it satisfies the property P. As a more complex record type example we use the following (simplified) version of IsCommutativeMonoid from Algebra.Structures in the standard library. The record type is parametrised over a carrier set, a relation, a binary operation and its identity element:

```
record IsCommutativeMonoid {A : Set} ( _≈_ : Rel A)
                                     ( _•_ : A → A → A) (ε : A) : Set₁ where
  field
    isSemigroup : IsSemigroup   _≈_   _•_
    identityˡ   : LeftIdentity  _≈_ ε _•_
    comm        : Commutative   _≈_   _•_
```

The fields capture the requirements of being a commutative monoid, in terms of three other properties. Here, the first field (isSemigroup) is also of record type; it is in fact common in Agda to define deeply nested record structures.

In Agda every record type also doubles as a module parametrised over a value of that type. For example, within the scope of the above record, given a value isNP : IsSemigroup {ℕ} _≈_ _+_, the phrase IsSemigroup isNP is meaningful in a context where Agda expects a module. It denotes a module containing a declaration for each field and helper definition of the IsSemigroup record type. Hence, within the scope of the above record, one can access the (nested) fields of isSemigroup in the module IsSemigroup isSemigroup. In

fact, it is very common for a record type to re-export all the definitions of inner records. This can be done with the following declaration (still inside the **record**):

```
open IsSemigroup isSemigroup public
```

open means that the new names are brought into scope for later definitions inside the record type (module) and **public** means the new names are also exported (publicly visible). This means that the user can ignore the nesting when fetching nested fields (but not when constructing them).

Equality proofs. We finish this section with an example of a helper definition that also serves as an introduction to *equality-proof notation*. A helper definition of IsCommutativeMonoid is the right identity proof $\mathsf{identity}^{\mathsf{r}}$, which is derivable from commutativity and left identity. We can define it as follows (still inside the **record** IsCommutativeMonoid):

```
identityʳ : ∀ x → (x • ε) ≈ x
identityʳ x =
  begin
    x • ε
  ≈⟨ comm x ε ⟩
    ε • x
  ≈⟨ identityˡ x ⟩
    x
  ■
```

The above is merely the composition by transitivity of $(\mathsf{comm}\ \mathsf{x}\ \varepsilon)$ and $(\mathsf{identity}^{\mathsf{l}}\ \mathsf{x})$, but by using special purpose operators the user can keep track of the intermediate steps in the proof in a style close to pen-and-paper proofs.

3. Formal Development Overview

In the following sections we expose our formalisation of the specification of the transitive closure algorithm, its implementation (Valiant's algorithm) and the proof of correctness. The algorithm falls out from a calculational refinement of the specification rather than being exposed *ex nihilo* and proved separately. The development is presented in a number of stages:

- We define the ring-like algebraic structure where we set our development (section 4),
- We give the specification of the transitive closure (section 5),
- We define the concrete data structures that the algorithm manipulates (section 6),
- We derive Valiant's algorithm from part of that specification (section 7),
- We conclude by showing that the algorithm satisfies the rest of the specification (section 8).

In figure 2 we present the mapping between the order of presentation in the paper and in the Agda development. The paper starts from a simplified presentation of the development including the top level algorithm and only then builds up to include all the properties and proofs needed for the full formalization. This makes the proof easier to follow and lets us explain step by step the full algebra needed. The Agda code, on the other hand, introduces the full algebra earlier and only gets to the top level algorithm on the last line of the file.

Figure 2: Mapping from code sections to paper sections.

4. Algebra

We begin by defining a record type called SemiNearRing, whose fields and helper definitions capture the algebraic structure that we need for the algorithm development. First we introduce a carrier set s with an equivalence relation, a zero, addition and multiplication.

```
record SemiNearRing : Set₁ where    -- C1: SemiNearRing
  field                             -- C1.1: Carriers, operators
    s : Set
    _≃ₛ_ : s → s → Set
    0ₛ   : s
    _+ₛ_ : s → s → s
    _·ₛ_ : s → s → s
```

Sets of non-terminals form a SemiNearRing with finite sets of non-terminals for s and its operations for the other fields (the usual equality, the empty set as zero, set union as addition and "cross product filtered by the grammar" as multiplication).

Further, the charts (from section 1), also form a SemiNearRing. Indeed, lifting the operations on matrices preserve the SemiNearRing structure, as we formally prove in section 6.

At this stage they are "raw" operations without laws. Here is a summary of the laws needed. We require that (0,+) forms a commutative monoid. We also require that 0 is absorbing for $(\cdot)$, that $(\cdot)$ distributes over $(+)$ and that $(+)$ is idempotent. Note that the product is not necessarily associative; in fact, if it were, computing the transitive closure would be much easier and parsing would not be an application.

$$x + 0 = x = 0 + x \qquad x + y = y + x \qquad x + (y + z) = (x + y) + z \qquad \text{(now)}$$
$$x \cdot 0 = 0 = 0 \cdot x \qquad x + x = x \qquad x \cdot (y + z) = x \cdot y + x \cdot z \qquad \text{(later)}$$

(It is easy to check that the definitions given in section 1 have these properties.) We could specify these properties individually, but instead we take advantage of "specification building blocks" from the Algebra modules in the Agda standard library [Danielsson and The Agda Team, 2013]. More specifically we use the record IsCommutativeMonoid from the library module Structures and we use the left and right zero laws from the parametrised module FunctionProperties specialised to the underlying equivalence ($\_\simeq_s\_$) on our carrier set s.

```
open Algebra.Structures                    using (IsCommutativeMonoid)
open Algebra.FunctionProperties _≃ˢ_ using (LeftZero; RightZero)
```

Armed with these properties we now continue the SemiNearRing record type by specifying (as new fields in the record) the laws we require of our operations: $\_+_s\_$ is a commutative monoid with 0_s as the unit, 0_s is also a multiplicative zero of $\_\cdot_s\_$ and multiplication preserves equivalence.

```
field                                      -- C1.2: Commutative monoid (+, 0)
  isCommMon : IsCommutativeMonoid _≃ˢ_ _+ˢ_ 0ˢ
  zeroˡ     : LeftZero  0ˢ _·ˢ_           -- expands to ∀ x → (0ˢ ·ˢ x) ≃ˢ 0ˢ
  zeroʳ     : RightZero 0ˢ _·ˢ_           -- expands to ∀ x → (x ·ˢ 0ˢ) ≃ˢ 0ˢ
  _<·>_      : ∀ {x y u v} → (x ≃ˢ y) → (u ≃ˢ v) → (x ·ˢ u ≃ˢ y ·ˢ v)
```

The rest of the record type consists of helper declarations which are useful to have in scope when working with the specification. They will be put in scope whenever we access an instance of the record in the definition of the algorithm or in its proof. Also inside the record type we specify the precedence of operators using the following declarations:

```
infix 4 _≃ˢ_; infixl 6 _+ˢ_; infixl 7 _·ˢ_
```

Exporting inner names. As we mentioned in section 2 the module Algebra.Structures includes a record (IsCommutativeMonoid) which contains the commutative monoid laws, and doubles up as a parametrised module. In the SemiNearRing record type that we are defining there is already a field isCommMon which, in turn, contains the proofs of the monoid laws so that a user with a value snr : SemiNearRing can access all these proofs by indexing through the two record layers. But two (and later more) levels of records is inconvenient to use so we include short hand names for the inner record fields as follows.

```
                                        -- C1.4: Exporting commutative monoid operations
open Algebra.Structures.IsCommutativeMonoid isCommMon public
  hiding (refl)
  renaming
    (isEquivalence to isEquiv_s
    ; assoc        to assoc_s
    ; comm         to comm_s
    ; •-cong       to _<+>_
    ; identity^l   to identity^l_s
    )
identity^r_s = proj_2 identity
```

Recall that **open** means that the new names are brought into scope for later definitions inside the record type and **public** means the new names are also exported (publicly visible). We rename (with a subscript s-suffix) to avoid clashes when we add most of the same for another set u later. The infix notation _<+>_ for congruence is useful in equality proofs: if we have proofs ap : a $\simeq_s$ a′ and bp : b $\simeq_s$ b′ we get a proof of (a $+_s$ b) $\simeq_s$ (a′ $+_s$ b′) which can be layed out nicely:

```
begin
    a      +_s  b
≈⟨ ap      <+>  bp ⟩
    a′     +_s  b′
■
```

Finally sSetoid packages up the carrier set s, the relation _$\simeq_s$_ and the proof that it is an equivalence. This is useful not only for documentation purposes (s *is* a setoid), but certain parts of the standard library require properties to be packaged in such a manner, for example the module giving convenient syntax for equality proofs, which we use later. We write 0_L for universe level 0.

```
sSetoid : Setoid 0_L 0_L                -- C1.5: Setoid, ...
sSetoid = record {Carrier       = s;
                  _≈_           = _≃_s_;
                  isEquivalence = isEquiv_s}


open IsEquivalence isEquiv_s public
  hiding (reflexive) renaming (refl to refl_s; sym to sym_s; trans to trans_s)
```

The **open public** of isEquiv$_s$ is (again) to avoid the need for multiple layers of record projections.

To summarise, a record value snr : SemiNearRing contains as fields a carrier set s, the operations (_$\simeq_s$_, 0_s, _$+_s$_, _$\cdot_s$_) and the proofs of the properties (isCommMon, zero^l, zero^r, _<·>_). We will later return to this record type and add a few more fields (in section 5) and helpers (in section 7.2 and section 8) to capture the full specification of the closure algorithm, but for now this will do.

4.1. **Matrix Algebra(s).** Valiant's algorithm works on square matrices. We carry on and define the algebraic structures required on matrices for Valiant's algorithm to work. Some of the structure of these matrices is the same as that required of elements, and we will later (in section 6) show that the additional properties follow from just a SemiNearRing structure on the elements.

The name of the carrier set (s) defined earlier was in reference to square matrices. However, Valiant's algorithm works only on (strictly) upper triangular matrices— that is, those whose elements at and below the diagonal are zero. We could have defined upper triangular matrices as the type of square matrices satisfying the triangularity predicate. However, such a definition yields a large amount of tedious work to manipulate the extra predicate.

Instead, we enforce the property of upper-triangularity axiomatically, by defining a separate type u (for upper triangular matrix) and require an embedding from u to s (square matrix). We package the types u and s, together with all their properties, in a single record SemiNearRing2. This packaging helps, because we later build structures by induction, and the inductive case for u depends on the induction hypothesis for s and *vice-versa*. (Yet we keep SemiNearRing as a separate entity, as it is an adequate specification of the elements of the matrices.)

```
record SemiNearRing2 : Set₁ where    -- C2: SemiNearRing2
  field
    snr : SemiNearRing
  open SemiNearRing snr public    -- public = export the "local" names from SemiNearRing
  field                                -- C2.1: Plus and times for u, ...
    u : Set
    _+ᵤ_ : u → u → u
    _·ᵤ_ : u → u → u
    u2s  : u → s
```

Beside the type u and an embedding into s, we require addition and multiplication over u: this is a simple specification of the property that those operations preserve upper-triangularity. Note that we require no relation between $\_+_u\_$ and $\_+_s\_$, no relation between $\_\cdot_u\_$ and $\_\cdot_s\_$ and no algebraic properties of the u-operations. Most of the link between u and s will become manifest when we see the recursive structure tying them together. The rest of the record contains helper definitions to lift the structure of s onto u using u2s. The lifting of binary relations is provided for us by the standard library via the infix operator Function.on.

```
  _≃ᵤ_ : u → u → Set
  _≃ᵤ_ = _≃ₛ_ Function.on u2s

  _u*s_ : u → s → s
  _u*s_ u s = u2s u ·ₛ s

  _s*u_ : s → u → s
  _s*u_ s u = s ·ₛ u2s u
```

The operator precedences and a Setoid instance follow, for completeness.

```
monoTimesLeft : ∀ {a} {b} {c} → (a ≤s b) → ((a ·s c) ≤s (b ·s c))
monoTimesLeft {a} {b} {c} a≤b =
  begin
    (a ·s c) +s (b ·s c)
  ≈⟨ sym (distr _ _ _) ⟩
    (a +s b) ·s c
  ≈⟨ a≤b <·> reflₛ ⟩    -- a ≤ b means a + b = b
    b ·s c
  ■
```

Figure 3: Monotonicity for multiplication follows from distributivity.

```
infix 4 _≃u_; infixl 6 _+u_; infixl 7 _·u_ _u·s_ _s·u_
uSetoid : Setoid 0ₗ 0ₗ
uSetoid = record {isEquivalence = Relation.Binary.On.isEquivalence u2s isEquivₛ}
```

To summarise, a record value snr2 : SemiNearRing2 contains the s-operations from a subrecord snr, the corresponding operations for u (but no laws) and the embedding u2s : u → s.

5. Specification

Recall that Valiant's algorithm aims at finding the smallest matrix C such that $C = W + C \cdot C$. Our goal is to find a function computing C, and to prove it correct. The first step is to formally transcribe the definition of transitive closure. Hence we revisit and extend the SemiNearRing structure with a few more components needed to define the closure relation. First we need an order to define "smallest". Remember that, for parsing, a value of type s is a set of non-terminals, or a matrix thereof. Hence, parsing uses set inclusion for the preorder and set union for addition and in this context a ⊆ b iff a ∪ b = b. A natural generalisation is to define a partial order where x ≤s y iff x +s y ≃s y. For the relation _≤s_ to be a preorder we need idempotence of addition (which implies reflexivity and transitivity). With this order addition is automatically monotonous, but for multiplication to be monotonous we need distribution laws: that _·s_ distributes over _+s_ (see figure 3).

```
open Algebra.FunctionProperties _≃s_
  using (Idempotent; _DistributesOverˡ_; _DistributesOverʳ_)
field                                   -- C1.3: Distributive, idempotent, ...
  idem    : Idempotent _+s_
  distl   : _·s_ DistributesOverˡ _+s_
  distr   : _·s_ DistributesOverʳ _+s_
      -- expands to ∀ a b c → (a +s b) ·s c ≃s (a ·s c) +s (b ·s c)
infix 4 _≤s_
_≤s_ : s → s → Set
x ≤s y = x +s y ≃s y
```

The algebraic structure we have specified so far in this record type is nearly, but not quite, an idempotent semiring. We just lack (and don't want) associativity (and unit) of our

multiplication. (Remember that for our motivating example, parsing, the multiplication operation is normally not associative.)

As the last addition to the SemiNearRing record type we define lower bound with respect to this order:

```
LowerBound_s = LowerBound _≤s_   -- C1.6: Lower bounds
```

In SemiNearRing2 we obtain a suitable ordering on u by lifting the ordering on s.

```
_≤u_ : u → u → Set
_≤u_ = _≤s_ Function.on u2s
```

Closure. We have now seen the first layer SemiNearRing specifying the underlying carrier set s for square matrices and the second layer SemiNearRing2 specifying the set u for upper triangular matrices. The third layer ClosedSemiNearRing specifies the transitive closure and Valiant's algorithm for computing it.

```
record ClosedSemiNearRing : Set1 where   -- C3: ClosedSemiNearRing
  field
    snr2 : SemiNearRing2   -- includes s, u and corresponding operations
  open SemiNearRing2 snr2
```

We can now finally give the specification and we do it in three steps: the relation Q capturing the "quadratic" equation, the relation Closure capturing only the "least" solutions of Q and finally Entire Q which says that there is a total function inside the relation.

```
  Q : u → u → Set                                -- C3.1: Quadratic equation Q + properties
  Q w c = w +u c ·u c ≃u c
  Closure : u → u → Set
  Closure w c = Least _≤u_ (Q w) c
  field
    entireQ : Entire Q
```

From the entireQ field (which we will populate later) we can extract the closure algorithm (closure) and part of its correctness proof (closureHasAll).

```
  closure : u → u                                -- C3.2: Closure function and correctness
  closure = fun entireQ
  closureHasAll : ∀ {w : u} → Q w (closure w)
  closureHasAll = correct entireQ
```

Proving Entire Closure means to show that closure yields the smallest possible solution, and is deferred until section 8.

```
  open SemiNearRing2 snr2 public
```

6. Matrices

In the specification and implementation we make heavy use of square matrices of size $\mathsf{m} \times \mathsf{m}$ for different values of m. For concision we make the simplifying assumption that m is a power of two ($\mathsf{m} = 2^{\mathsf{n}}$) — handling the general case involves indexing matrices with their sizes, which is straightforward but clutters the development (section 10.1).

There are many different ways to represent matrices. As usual when working with dependent types, some definitions yield a concise presentation, while others require large amounts of boilerplate, obscuring the intent. A judicious choice is thus in order. Possibilities include:

- A vector of vectors of elements, where a vector is

```
data Vec a n where
  Nil  : Vec a 0
  Cons : a → Vec a n → Vec a n → Vec a (1 + n)
```

- A function from (two) indices to elements `Bin n → Bin n → a` (where `Bin` stands for a binary number of `n` bits)
- A recursive data-type (a quad tree):

```
data Mat a n where
  Unit : a → Mat a 0
  Quad : Mat a n → Mat a n → Mat a n → Mat a n → Mat a (1 + n)
```

- A function from `n` to `Set`:

```
Mat a 0       = OneByOne a
  where OneByOne a = a
Mat a (suc n) = Square (Mat a n)
  where Square t = t × t × t × t
```

Here we choose the last approach, which is specifically tailored to the problem at hand. Indeed, the advantage of doing so is that `Square` is a functor. This approach allows to extend the `Square` functor with all the axiomatisation we need, including ring-like structures and up to the full closure specification. In fact, most of the rest of the paper is devoted to the definition of that functor: when it is fully defined we have a proven-correct implementation of Valiant's algorithm. In this section we show the abstraction over `SemiNearRing` and the construction of matrices, up to the full definition of `Mat`. In the base case `OneByOne` (shown later in section 6 on page 17) we lift a `SemiNearRing` to a `ClosedSemiNearRing` and in the inductive case and we apply `Square` which preserves the `ClosedSemiNearRing` structure.

The main functor: `Square`. We attack the recursive case right away by defining the function `Square` which lifts our algebraic structure (the types `s` and `u`, the operations on them and all the laws) from elements to 2-by-2 (block) matrices. It can be seen as an implementation of a (first-class, but otherwise) ML-style functor.

```
  -- C4: 2-by-2 block matrix, preserving ClosedSemiNearRing
Square : ClosedSemiNearRing → ClosedSemiNearRing
Square csnr = CSNR where
  open ClosedSemiNearRing csnr
```

The rest of this section is the body of this **where** clause where all operations and laws from csnr are in scope.

Lifting types. We start by defining two types U and S, instantiating the u and s types for two-by-two (block) matrices. Throughout the rest of the paper, we use the convention of using capitals for the values of the fields of the record created (the target of the functor).

For type S we could use $\mathsf{S} = \mathsf{s} \times \mathsf{s} \times \mathsf{s} \times \mathsf{s}$ but we prefer to use a record type for clarity. The constructor name uses angle brackets to lift any ambiguity.

```
record S : Set where                    -- C4.1: Square matrix
  constructor ⟨_,_,_,_⟩
  field
    s00 : s;   s01 : s
    s10 : s;   s11 : s
infix 4 ⟨_,_,_,_⟩
```

The two-by-two upper triangular matrix is composed of two (smaller) upper-triangular matrices, and a square matrix for the top-right corner. (The bottom-left corner is zero.) Basically, $\mathsf{U} = \mathsf{u} \times \mathsf{s} \times \mathsf{u}$ but again, regular nested pairs obscure the intent. Here we add a box inside the name of the record constructor, as a reminder that the bottom left corner is empty.

```
record U : Set where                    -- C4.2: Upper triangular matrix
  constructor ⟨_,_,•,_⟩
  field
    uu00 : u;   us01 : s;
                uu11 : u
infix 4 ⟨_,_,•,_⟩
```

Lifting operations. We lift the operations $0, +, \cdot$ from the underlying semi-near-ring to matrices over that semi-near-ring, in the usual manner.

```
_+S_ : S → S → S
_+S_ ⟨ a  , b  , c  , d  ⟩
     ⟨ a' , b' , c' , d' ⟩ =
     ⟨ a +s a' , b +s b' , c +s c' , d +s d' ⟩
```

Matrix multiplication is defined as expected.

```
  _·S_ : S → S → S
  _·S_ ⟨ a , b ,
         c , d ⟩ ⟨ a' , b' ,
                   c' , d' ⟩ = ⟨ (a ·s a') +s (b ·s c') , (a ·s b') +s (b ·s d')
                               , (c ·s a') +s (d ·s c') , (c ·s b') +s (d ·s d') ⟩
  infixl 6 _+S_
  infixl 7 _·S_
```

Zero is defined similarly.

```
zerS : S
zerS = ⟨ 0ₛ , 0ₛ ,
         0ₛ , 0ₛ ⟩
```

We can then define the operations on upper-triangular matrices. In particular, we give full definitions of point-wise upper triangular matrix addition and multiplication. Having to define operations both for U and S is an consequence of our choice of separating u and s in the algebraic structure. This may look redundant, but in fact the definitions for U serve the purpose of proving that upper-triangularity is preserved by those operations. In particular, not that u-operations are used on the (block-) diagonal and s-operations are used in the the "square corner".

```
_+ᵁ_ : U → U → U
_+ᵁ_ ⟨ xl , xm , •, xr ⟩ ⟨ yl , ym , •, yr ⟩ = ⟨ xl +ᵤ yl , xm +ₛ ym , •, xr +ᵤ yr ⟩
_·ᵁ_ : U → U → U
_·ᵁ_ ⟨ xl , xm
       , •, xr ⟩ ⟨ yl ,  ym
                 , •,  yr ⟩  = ⟨ xl ·ᵤ yl ,   xl ᵤ·ₛ ym +ₛ xm ₛ·ᵤ yr
                               , •,           xr ·ᵤ yr ⟩
```

Equivalence is structural: two matrices are equivalent if all the elements are equivalent.

```
_≃ˢ_ : S → S → Set
_≃ˢ_ ⟨ a , b , c , d ⟩ ⟨ a′ , b′ , c′ , d′ ⟩ = (a ≃ₛ a′) × (b ≃ₛ b′)
                                            × (c ≃ₛ c′) × (d ≃ₛ d′)
infix 4 _≃ˢ_
```

Embedding upper triangular matrices as (regular) matrices is straightforward:

```
U2S : U → S
U2S ⟨ uu₀₀ , us₀₁ , •, uu₁₁ ⟩ = ⟨ u2s uu₀₀ ,   us₀₁ ,
                                  0ₛ        ,   u2s uu₁₁ ⟩
```

Lifting laws. At this point we must verify that the above constructions preserve the laws seen so far. Most of this verification (of symS, transS, congS, etc.) is by straightforward, and omitted, lifting of the proofs from the underlying structure, for example reflS:

```
reflS : {x : S} → x ≃ˢ x                    -- C4.3: Laws
reflS = reflₛ , reflₛ ,
        reflₛ , reflₛ
```

To prove the preservation of distributivity we use the following lemma about commutativity (proving it is an easy equational reasoning exercise in Agda).

```
swapMid : ∀ {a b c d} →  (a +ₛ b) +ₛ (c +ₛ d)
                      ≃ₛ (a +ₛ c) +ₛ (b +ₛ d)

distlS : (x y z : S) → x ·ˢ (y +ˢ z) ≃ˢ x ·ˢ y +ˢ x ·ˢ z
distlS _ _ _ = distrHelp , distrHelp , distrHelp , distrHelp
```

```
  where distrHelp : ∀ {a b c d e f} →  a ·s (b +s c)      +s d ·s (e +s f)
                                    ≃s (a ·s b +s d ·s e) +s (a ·s c +s d ·s f)
        distrHelp = trans_s (distl _ _ _ <+> distl _ _ _) swapMid
```

The record value of type Square should eventually contain all the fields, including the proof of closure — but we leave the rest to the upcoming sections. Thus we leave the definition of the functor Square for now and instead give the the base case of the inductive structure.

Base case: 1-by-1 matrices. Any strictly upper triangular matrix is by definition zero on (and below) the diagonal. At size 1-by-1 that means the only element is zero so need not store any element. We represent this case by the unit type (tt : ⊤) and it is therefore trivially equipped with closure.

```
OneByOne : SemiNearRing → ClosedSemiNearRing   -- C5: One-by-one matrix
OneByOne snr = record {snr2 = record {snr = snr; u = ⊤; _+_u_ = λ _ _ → tt;
                                      u2s = λ _ → 0s;   _·u_ = λ _ _ → tt}
                       }
  where open SemiNearRing snr using (0s)
```

Finally we give the recursion schema, following the pattern that we hinted at the beginning of the section: we define Mat n as the semi-near-ring structures lifted n + 1 times.

```
Mat : ℕ → SemiNearRing → ClosedSemiNearRing   -- C6: Top level recursion for matrices
Mat zero    el = OneByOne el
Mat (suc n) el = Square (Mat n el)
```

The fields of Mat n el are of special interest to us. In particular the u field is the type of (strictly upper triangular) matrices of size 2^n, which we will soon show how to equip with a closure operation.

```
Upper : ℕ → SemiNearRing → Set
Upper n el = ClosedSemiNearRing.u (Mat n el)
```

(The expression ClosedSemiNearRing.u csnr extracts the field u from the record csnr of type ClosedSemiNearRing.)

7. Transitive closure: derivation

We have now completed the definition of all the structures necessary to develop our proof. In this section we describe (and partially prove) the closure algorithm.

We do so by deriving it from the specification Closure w c = Least _≤u_ (Q w) c, where Q w c = w +u c ·u c ≃u c and we start with the equational part, the Entire Q requirement, and return to Least in section 8. Technically, we give a definition for the entireQ : Entire Q field in both the OneByOne and Square cases of the Mat function.

We first proceed semi-formally, to show what a non-certified proof of the algorithm looks like, and to be able to compare it with the fully formal, certified Agda proof that we subsequently present.

7.1. **Closure of triangular matrices.** Recall that our task is to find a function $\_^+$ which maps an upper triangular matrix W to its transitive closure $C = W^+$.

If W is a 1 by 1 matrix, $C = W = 0$. Otherwise, let us divide W and C in blocks as follows:

$$W = \begin{bmatrix} A & Y \\ 0 & B \end{bmatrix} \qquad\qquad C = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix}$$

Then the condition that C satisfies Q becomes:

$$\begin{bmatrix} A & Y \\ 0 & B \end{bmatrix} + \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} \cdot \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix}$$

Applying matrix multiplication and addition block-wise:

$$\begin{aligned} A + A'A' &= A' \\ Y + A'X' + X'B' &= X' \\ B + B'B' &= B' \end{aligned}$$

Because A and B are smaller matrices than W (and still upper triangular), we know how to compute A' and B' recursively ($A' = A^+$, $B' = B^+$). Before showing how X' is computed, we show how to formalise the above reasoning in Agda. The main job is to populate the entireQ field in the ClosedSemiNearRing record. We have to do so both for the base case and the recursive case of the Mat construction. The base case being trivial, we show here the inductive case.

We first define the Q relation for 2-by-2 (block-)matrices:

```
QU = λ W C → (W +U (C ·U C)) ≃U C  -- C4.4: Lifting Q and its proof
```

and then we proceed with the proof that it is Entire: for any input matrix W we construct another matrix C and a proof that it is a closure (strictly speaking, so far only that it satisfies QU W C). One remarkable feature is that Agda can infer the solution matrix (C = _), from the proof. The proof follows exactly the semi-formal development given at the beginning of the subsection.

```
entireQStep : ∀ W → ∃ (QU W)
entireQStep W = C , proof where
  C : U
  C = _
  open EqReasoning (SemiNearRing2.uSetoid SNR2)
  proof : (W +U (C ·U C)) ≃U C
  proof = begin
            (W +U (C ·U C))
          ≡⟨ refl ⟩  -- expand matrix components
            let ⟨ A , Y , •, B ⟩ = W
                ⟨ A' , Y' , •, B' ⟩ = C in
            ⟨ A , Y , •, B ⟩ +U (⟨ A' , Y' , •, B' ⟩ ·U ⟨ A' , Y' , •, B' ⟩)
          ≡⟨ refl ⟩  -- expand definition of ·U
            ⟨ A , Y , •, B ⟩ +U ⟨ A' ·u A' , (A' u*s Y' +s Y' s*u B')
                                , •, B' ·u B' ⟩
          ≡⟨ refl ⟩  -- by def. of +U
```

```
  ⟨ A +u A′ ·u A′   ,   Y +s (A′ u·s Y′ +s Y′ s·u B′)
                        , •, B +u B′ ·u B′ ⟩
≈⟨ congU closureHasAll completionHasAll closureHasAll ⟩
  ⟨ A′ , Y′ , •, B′ ⟩
≡⟨ refl ⟩
  C
∎
```

The definition `C = _` works because Agda unifies the type of the proof that it infers with the type that we give ((W +U (C ·U C)) ≃U C). The left-hand-side of the equivalence inferred type is given by each step in the proof: `C = ⟨ A′ , Y′ , •, B′ ⟩`, and so on. If we expand those steps this is the core algorithm:

```
C = let  ⟨ A , Y , •, B ⟩ = W
         (A′ , proofA)    = entireQ A
         (B′ , proofB)    = entireQ B
         (Y′ , proofY)    = entireL A′ Y B′
     in  ⟨ A′ , Y′ , •, B′ ⟩
```

The use of equational reasoning in proof shows another very useful feature of the proof notation: using a normal **let-in** expression together with the distfix transitivity operator _≡⟨_⟩_ we can do what is often done in paper-proofs: introduce new names in the middle of the reasoning chain. The new names (here A, Y, B etc.) are in scope in the rest of the proof. Another interesting feature of this proof is that many of the steps can be justified simply by refl : x ≡ x. Indeed, Agda automatically expands definitions during type-checking, and thus automatically expands the definitions of operators on block matrices. In fact, because refl is "the unit of transitivity", Agda would be just as happy with only:

```
proof = congU closureHasAll completionHasAll closureHasAll
```

Yet, this one line still holds the full proof of the inner induction (completionHasAll) which is the topic of section 7.2. (Remember that closureHasAll = correct entireQ from section 5 (page 13) plays the role of induction hypothesis for the closure of triangular matrices.)

The base case is completely trivial; a 1-by-1 upper triangular matrix contains no non-zero element and is represented by the unit type. Thus entireQBase tt = tt , refl$_s$.

7.2. **Completion of square matrices.** As in the previous subsection, we first proceed semi-formally. The problem is to find a recursive function V which maps A, Y and B to $X = V(A, Y, B)$, such that $X = Y + A \cdot X + X \cdot B$. In terms of parsing, the function V combines the chart A of the first part of the input with the chart B of the second part of the input, via a *partial* chart Y concerned only with strings starting in A and ending in B, and produces a full chart X. We proceed as before and divide each matrix into blocks:

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \quad Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \quad A = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$$

The condition on X then becomes

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \cdot \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} + \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$$

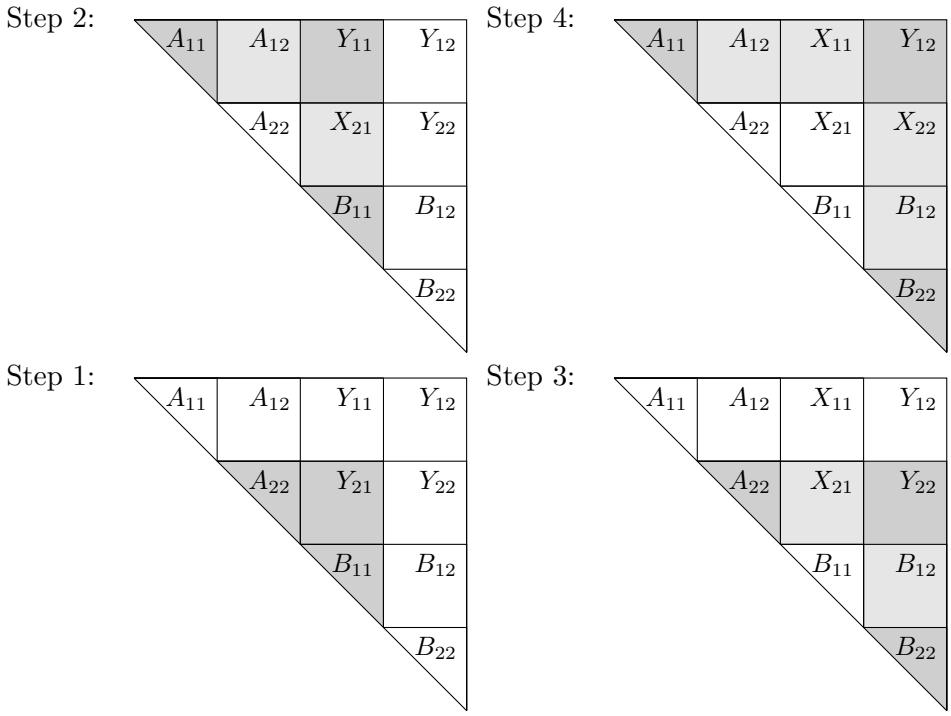

Figure 4: The recursive step of function V. The charts A and B are already complete. To complete the matrix Y, that is, compute $X = V(A, Y, B)$, one splits the matrices and performs 4 recursive calls. Each recursive call is depicted graphically. In each figure, to complete the dark-gray square, multiply the light-gray rectangles and add them to the dark-gray square, then do a recursive call on triangular matrix composed of the completed dark-gray square and the triangles.

By applying matrix multiplication and addition block-wise:

$$\begin{aligned}
X_{11} &= Y_{11} + A_{11}X_{11} + A_{12}X_{21} + X_{11}B_{11} + 0 \\
X_{12} &= Y_{12} + A_{11}X_{12} + A_{12}X_{22} + X_{11}B_{12} + X_{12}B_{22} \\
X_{21} &= Y_{21} + 0 + A_{22}X_{21} + X_{21}B_{11} + 0 \\
X_{22} &= Y_{22} + 0 + A_{22}X_{22} + X_{21}B_{12} + X_{22}B_{22}
\end{aligned}$$

By commutativity of $(+)$ and 0 being its unit:

$$\begin{aligned}
X_{11} &= Y_{11} + A_{12}X_{21} + A_{11}X_{11} + X_{11}B_{11} \\
X_{12} &= Y_{12} + A_{12}X_{22} + X_{11}B_{12} + A_{11}X_{12} + X_{12}B_{22} \\
X_{21} &= Y_{21} + A_{22}X_{21} + X_{21}B_{11} \\
X_{22} &= Y_{22} + X_{21}B_{12} + A_{22}X_{22} + X_{22}B_{22}
\end{aligned}$$

Now we have four equations, all of the form that V can compute solutions to. Because each of the sub-matrices is smaller and because of the absence of circular dependencies, X can be computed recursively by V. The internal dependencies dictate the order: start computing

X_{21}, use that to compute X_{11} and X_{22} (possibly in parallel) and finally compute X_{12}:

$$\begin{aligned}
X_{21} &= V(A_{22},\, Y_{21} &&,\, B_{11})\\
X_{11} &= V(A_{11},\, Y_{11} + A_{12}X_{21} &&,\, B_{11})\\
X_{22} &= V(A_{22},\, Y_{22} + X_{21}B_{12} &&,\, B_{22})\\
X_{12} &= V(A_{11},\, Y_{12} + A_{12}X_{22} + X_{11}B_{12},\, B_{22})
\end{aligned}$$

A graphical summary is shown in fig. 4.

We proceed to certify this proof step in Agda. The problem is to solve the equation L, defined as follows. (We pick the name L for linear equation as we used Q for quadratic earlier.)

```
L : u → s → u → s → Set                -- C2.2: Linear equation L
L a y b x = y +s (a u·s x +s x s·u b) ≃s x
```

We must prove that the relation L is entire, and thus get an algorithm (to compute the last argument) as well as its correctness proof. To this end, we add the appropriate field to the ClosedSemiNearRing record. We now reap the full benefits of using a record structure, which frees us from repeating the recursion pattern.

```
  field                                -- C3.3: Function for L and its correctness
    entireL : Entire3 L
  completion : u → s → u → s
  completion = fun3 entireL
  completionHasAll : ∀ {a y b} → L a y b (completion a y b)
  completionHasAll = correct3 entireL
```

Again, the bulk of the proof is the recursive case (part of the definition of Square), the lifting of completionHasAll to 2-by-2 block matrices. Here as well, the semi-formal proof is faithfully represented in Agda.

```
                                       -- C4.6: Lifting the proof of L
  entireLS : ∀ (A : U) (Y : S) (B : U) → ∃ (λ X → Y +S (A U·S X +S X S·U B) ≃S X)
  entireLS A Y B = X , proof where
    X : S
    X = _      -- filled in by unification with Y +S (U2S A ·S X +S X ·S U2S B)
    proof : Y +S (U2S A ·S X +S X ·S U2S B) ≃S X
    open EqReasoning (SemiNearRing2.sSetoid SNR2)
    proof =    -- continued below to fit the width of the paper (it is still in the where clause)

begin
  (Y +S (A U·S X +S X S·U B))
≡⟨ refl ⟩  -- name the components
  let ⟨ a00 , a01 , •, a11 ⟩   = A
      ⟨ b00 , b01 , •, b11 ⟩   = B
      ⟨ y00 , y01 , y10 , y11 ⟩ = Y
      ⟨ x00 , x01 , x10 , x11 ⟩ = X
  in Y +S (A U·S X +S X S·U B)
```

```
≡⟨ refl ⟩  -- expand _U·S and _S·U and use components
  let A·X = ⟨ a00 u's x00 +s a01 ·s x10 , a00 u's x01 +s a01 ·s x11
            , 0s ·s x00   +s a11 u's x10 , 0s ·s x01   +s a11 u's x11 ⟩
      X·B = ⟨ x00 s'u b00 +s x01 ·s 0s   , x00 ·s b01   +s x01 s'u b11
            , x10 s'u b00 +s x11 ·s 0s   , x10 ·s b01   +s x11 s'u b11 ⟩
  in Y +S (A·X +S X·B)

≡⟨ refl ⟩  -- Expand ≃S, +S and collect components
  ⟨ y00 +s ((a00 u's x00 +s a01 ·s x10)  +s (x00 s'u b00 +s x01 ·s  0s))
  , y01 +s ((a00 u's x01 +s a01 ·s x11)  +s (x00 ·s b01  +s x01 s'u b11))
  , y10 +s ((0s  ·s  x00 +s a11 u's x10) +s (x10 s'u b00 +s x11 ·s  0s))
  , y11 +s ((0s  ·s  x01 +s a11 u's x11) +s (x10 ·s b01  +s x11 s'u b11)) ⟩

≈⟨ congS zeroLemma00 zeroLemma01 zeroLemma10 zeroLemma11 ⟩
    -- assoc. and comm. of +; zero absorption.
  ⟨ y00 +s a01 ·s x10 +s                 (a00 u's x00 +s x00 s'u b00)
  , y01 +s a01 ·s x11 +s x00 ·s b01 +s (a00 u's x01 +s x01 s'u b11)
  , y10 +s                               (a11 u's x10 +s x10 s'u b00)
  , y11 +s x10 ·s b01 +s                 (a11 u's x11 +s x11 s'u b11) ⟩

≈⟨ congS completionHasAll completionHasAll completionHasAll completionHasAll ⟩
  ⟨ x00 , x01 , x10 , x11 ⟩
≡⟨ refl ⟩
  X
■
```

The series of zeroLemmas use that zeros are absorbing, in addition to commutativity and associativity of addition. One more time, the actual solution of the equation X can be inferred by Agda on the basis of the proof. The base case is a mere application of the properties of zero.

```
open import ZeroLemmas snr  -- C5.1: Base case for L
entireLBase : (a : ⊤) (y : s) (b : ⊤) → ∃ (λ x → y +s (a u's x +s x s'u b) ≃s x)
entireLBase tt y tt = y , zeroˡʳLemma y y y
```

We have now completely specified Valiant's algorithm, and it can be accessed via the appropriate field of Mat n:

```
-- C6.1: Top level algorithm extraction
valiantAlgorithm : (el : SemiNearRing) → ∀ n → Upper n el → Upper n el
valiantAlgorithm el n u = ClosedSemiNearRing.closure (Mat n el) u
```

8. SMALLEST

Only one bit of proof remains to obtain full correctness: namely, we should prove that the solution computed by our algorithm is a lower bound of all solutions of the Q equation. The proof is as follows. We have three lemmas. The first lemma is that the L equation (recall L a y b x = y $+_s$ (a $_{u\cdot s}$ x $+_s$ x $_{s\cdot u}$ b) $\simeq_s$ x) admits a single solution for x. (This is not surprising, as it is a linear equation.) Being unique, this solution is thus necessarily the smallest. The second lemma is that the L relation is a congruence in its second argument. The third lemma is that the completion function is monotonous. The theorem (lower bound) and the two lemmas are proved by induction, as before.

We start by stating the two first lemmas: 1. the relation L admits a unique solution in its last argument and 2. the relation L is a congruence in its second argument.

```
  -- C2.3: Properties of L
UniqueL = ∀ {a y b} → UniqueSolution _≃s_ (L a y b)
CongL   = ∀ {a x b} → ∀ {y y′} → y ≃s y′ → L a y b x → L a y′ b x
```

We then prove those two lemmas, as well as monotonicity of completion and the main theorem of this section (closureIsLeast), and we do so by induction on the matrix structure. Formally, we proceed as before: the induction pattern is encoded by adding fields to our ClosedSemiNearRing record. The fields to add are as follows:

```
  field                                   -- C3.4: Ordering properties of L and Q
    uniqueL        : UniqueL
    congL          : CongL
    completionMono : ∀ {a a′ y y′ b b′} → a ≤u a′ → y ≤s y′ → b ≤u b′ →
                                          completion a y b ≤s completion a′ y′ b′
    closureIsLeast : {w : u} → LowerBound _≤u_ (Q w) (closure w)
```

We then formalize our above remark: the uniqueness of L immediately implies that completion gives a least solution. We do this in the ClosedSemiNearRing structure, in order to get this result for every induction step.

```
  open OrderLemmas snr public

  completionIsLeast : ∀ (a : u) (y : s) (b : u) → LowerBound_s (L a y b) (completion a y b)
  completionIsLeast a y b z p = ≃sTo≤s (uniqueL completionHasAll p)
```

We then proceed with the induction proofs. First, we prove the induction case of our main theorem, from the induction hypotheses. As usual, the upper-right corner is the difficult case, and requires the monotonicity lemma.

```
_≤U_ = SemiNearRing2._≤u_ SNR2        -- C4.5: Lifting orders and their properties
_≤S_ = SemiNearRing2._≤s_ SNR2
closureIsLeastS : ∀ {W} → LowerBound _≤U_ (QU W) (fun entireQStep W)
closureIsLeastS Z QUWZ =
  let ⟨ z00 , z01 , •,  z11 ⟩ = Z          -- every matrix Z
      (p00 , p01 , _ , p11) = QUWZ         -- which satisfies (QU W Z)
      q10 = identity′_s 0s
      q00 = closureIsLeast z00 p00         -- is bigger than C = fun entireQStep W
      q11 = closureIsLeast z11 p11
```

```
    vs01      = completionIsLeast _ _ _ z01 p01
    mono01    = completionMono q00 (≃sTo≤s reflₛ) q11
  in ( q00  , ≤−transₛ mono01 vs01
     , q10  , q11)
```

We can then prove the induction case of the uniqueness of L. The proof uses the induction hypotheses, replicating the structure derived in the previous section. The only slight difficulty is to use the identity of 0 at the appropriate places.

```
uniqueLS : SemiNearRing2.UniqueL SNR2   -- C4.7: Proofs for ordering L-solutions
uniqueLS (p00 , p01 , p10 , p11) (q00 , q01 , q10 , q11)  =  eq00 , eq01 , eq10 , eq11
  where mutual
    s00  =  congL (reflₛ <+> (reflₛ <·> sym eq10))
    s11  =  congL (reflₛ <+> (sym eq10 <·> reflₛ))
    s01  =  congL ((reflₛ <+> (reflₛ <·> sym eq11)) <+> (sym eq00 <·> reflₛ))
    r10  =       trans (sym zeroLemma10) q10
    r00  =  s00 (trans (sym zeroLemma00) q00)
    r11  =  s11 (trans (sym zeroLemma11) q11)
    r01  =  s01 (trans (sym zeroLemma01) q01)
    eq10 =  uniqueL (trans (sym zeroLemma10) p10) r10
    eq00 =  uniqueL (trans (sym zeroLemma00) p00) r00
    eq11 =  uniqueL (trans (sym zeroLemma11) p11) r11
    eq01 =  uniqueL (trans (sym zeroLemma01) p01) r01
```

Completion monotonicity (induction case) is also straightforward, as is the base case. (The operators $[+]$ and $[*]$ are monotonicity for sum and product.)

```
completionMonoS : ∀ {a a′ y y′ b b′} → a  ≤U  a′ → y  ≤S  y′ → b  ≤U  b′ →
                    proj₁ (entireLS a y b)  ≤S  proj₁ (entireLS a′ y′ b′)
completionMonoS (p00 , p01 , p10 , p11) (q00 , q01 , q10 , q11)
                (r00 , r01 , r10 , r11)  =  m00 , m01 , m10 , m11
  where m10  =  completionMono p11 q10 r00
        m00  =  completionMono p00 (q00 [+] p01 [*] m10) r00
        m11  =  completionMono p11 (q11 [+] m10 [*] r01) r11
        m01  =  completionMono p00 (q01 [+] p01 [*] m11 [+] m00 [*] r01) r11

congLS : SemiNearRing2.CongL SNR2
congLS P Q  =  transS (symS P <+S> reflS) Q
```

```
CSNR : ClosedSemiNearRing
CSNR  =  record {
  snr2             =  SNR2;
  entireQ          =  entireQStep;
  closureIsLeast   =  closureIsLeastS;
  entireL          =  entireLS;
  uniqueL          =  uniqueLS;
  congL            =  congLS;
  completionMono   =  completionMonoS }
```

The base cases of lemmas are trivial and omitted. For the main theorem, we have:

```
leastQBase = λ _ _ → identity^I_s 0_s   -- C5.2: Base case for least Q
```

This concludes our proof. The complete development is checked by Agda 2.4, and is available as supplementary material for this paper. The set of Agda files is written in literate programming style, and doubles up as the input for the typesetting program.

9. Related Work

9.1. **Efficient Parsing.** One of the main motivations for this work is the discovery that Valiant's algorithm is not only interesting theoretically (as it gives an upper bound on the complexity of context-free recognition), but also practically.

Indeed Bernardy and Claessen [2013, 2015] have recently shown that the divide and conquer structure of Valiant's parsing algorithm yields an efficient parallel algorithm, given commonly occurring conditions on the input. In sum, if the input is organised hierarchically, then the conquer step is $O(\log^2 n)$, instead of being as complex as matrix multiplication.

9.2. **Certified Parsing.** Several certified parsers exist. Firsov and Uustalu [2013], Coquand and Siles [2011] have implemented parsers for regular languages. Jourdan et al. [2012] have implemented a parser for LR languages. Two certifications of full context-free parsers have been produced while the present paper was in submission.

Ridge [2014] has produced a novel, fully verified parser that also has good practical performance. Firsov and Uustalu [2014] have verified the CYK parsing algorithm (a precursor of Valiant's).

While Valiant's algorithm gives the best asymptotic bounds and is also known to behave well in many practical situations [Bernardy and Claessen, 2015], we have not measured the practical performance of (the Agda version of) our implementation. As we write, we expect it to be very bad: the program extraction mechanism of Agda is not mature and we cannot expect good results.

A previous formalisation of Valiant's algorithm was carried by Bååth Sjöblom [2013] under the supervision of the authors of this paper. The present work is a redevelopment of the proof from scratch. Indeed, the proof produced by Bååth Sjöblom is opaque: its structure does not match the informal proof. A close match between the formal and the informal proof was enabled by two key design choices: 1. our representation of matrices as an (extensible) record with all the necessary lemmas and 2. the use of two different types for square and upper triangular matrices (instead of using sigma types). In particular, using sigma types mean that every triangular matrix would be composed of two fields (a square matrix and a proof of triangularity). In turn, using this structure requires to writes a proof whenever such a matrix is produced. Baking triangularity into the structure of types avoids this complication.

9.3. **Certified Parsing Combinators.** In functional programming, one often uses parsing combinators as a language formalism. This means that the grammar is represented directly as code in the host language, instead of data structures. On the one hand, combinator parsing is generally less efficient than context-free parsing: the latter technique has access to the full grammar data, including its recursive structure, and thus more intelligent processing of the input is possible. On the other hand, combinators can in principle describe languages which are not context free.

Danielsson [2010] has formalised combinator parsing in Agda. The expressivity of Danielsson's formalism is maximal: it can express all languages which can be decided by an Agda algorithm.

While the parser of [Ridge, 2014], provides also a combinator interface, it begins by converting the grammar to a first order representation.

9.4. **Algebra of Programming.** The idea of deriving programs from specifications comes from the school of "Algebra of Programming" (AoP) [Bird and de Moor, 1997]. While AoP uses relational algebra, our setting is Agda. This paper is thus a direct descendent of "Algebra of Programming in Agda" [Mu et al., 2008, 2009], and as such leverages the power not only of Agda, but of an extensive set of standard libraries, developed over last decade mainly by Danielsson and The Agda Team [2013]. These libraries have already been used for several applications.

Yet, during the maturation of the present work we have found the AoP canon too limited for our purposes, and have departed from it significantly. We have not derived the algorithm from the full specification, as AoP prescribes, but only part of it (Equation Q).

Further, while the AoP school prescribes to perform derivations outside the recursive structure (and thus makes extensive usage of catamorphisms and their properties), we have first expanded the recursive structure, and used equational derivations for the inductive case. This primary unrolling of recursion significantly simplified our proof, as every theorem (and lemma) proved using the same induction pattern is simply added as a component of a record (coding up the induction hypothesis).

10. Extensions

10.1. **General size for matrices.** In order to deal with matrices of general sizes (not only $2^n \times 2^n$), we need to define the following type for the size of matrices:

```
data Shape : Set where
  Leaf : Shape                       -- One
  Bin : Shape → Shape → Shape        -- Sum of the two shapes
```

In addition to the size, the above type gives the way to divide a matrix into submatrices. It is a good idea to use the above structure instead of a (unary or binary) natural number, because the combination of sizes requires no computation.

Then, every type for matrices needs to be indexed on such a Shape, and in particular the functor generating matrices becomes a doubly-indexed functor, and the proof needs special cases when one of the dimensions is One.

```
data Mat : Shape → Shape → Set → Set where
  Quad : Mat x₁ y₁ a → Mat x₂ y₁ a →
         Mat x₁ y₂ a → Mat x₂ y₂ a →
         Mat (Bin x₁ x₂) (Bin y₁ y₂) a
  OneByOne : a → Mat Leaf Leaf a
  Row : Mat x₁ Leaf a → Mat x₂ Leaf a → Mat (Bin x₁ x₂) Leaf a
  Col : Mat Leaf y₁ a → Mat Leaf y₂ a → Mat Leaf (Bin y₁ y₂) a
```

10.2. **Boolean Grammars.** Okhotin [2014] has shown how to extend Valiant's algorithm to parse Boolean grammars. Boolean grammars allow to define the generation of non-terminals not only by union of production rules, but also intersection and complement. They can characterise non context-free languages, such as $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. In this case, the ring-like structure that we have used is not sufficient: one must apply a Boolean function to all possible combination of non-terminals before obtaining the parses of a given substring.

We believe that a straightforward extension of our proof to Okhotin's variant is possible.

10.3. **Semirings and relatives.** Our formalisation work showed that the specification could be generalised to work for (matrices over) arbitrary closed semirings [Dolan, 2013] and even further. Comparing to the theory explored in Dolan's paper they mention already on their page 1: "If we have an affine map $x \to ax + b$ in some closed semiring, then $x = a* b$ is a fixpoint". It appears that our linear equation `x = y + a x + x b` in section 7.2 is the natural generalisation of the affine map fixed point to the case of non-commutative algebra and that (the corner case `V` of) Valiant's algorithm computes this fixed point for upper triangular matrices. Future work includes exploring this relation in more detail and perhaps generalise out development to arbitrary (non-triangular) matrices.

10.4. **Sparse matrices.** The efficiency of Valiant's algorithm in the average case relies on using sparse matrices [Bernardy and Claessen, 2015]. The above proof does not deal with sparseness. Yet, it is straightforward to support sparseness as outlined by Bernardy and Claessen [2015]: one needs to change the `U` type to be a disjunction between the 2-by-2 case and the empty matrix case.

References

J.-P. Bernardy and K. Claessen. Efficient divide-and-conquer parsing of practical context-free languages. In *Proc. of ICFP 2013*, pages 111–122, 2013.

J.-P. Bernardy and K. Claessen. Efficient parallel and incremental parsing of practical context-free languages. *J. of Funct. Prog.*, 25, 2015. ISSN 1469-7653. doi: 10.1017/S0956796815000131.

R. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice-Hall International, 1997.

T. Bååth Sjöblom. *An Agda proof of the correctness of Valiant's algorithm for context free parsing*. MSc thesis, Chalmers University of Tech., 2013.

N. Chomsky. *Syntactic Structures*. Mouton de Gruyter, 1957.

N. Chomsky. On certain formal properties of grammars. *Information and control*, 2(2): 137–167, 1959.

T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In *Certified Programs and Proofs*, pages 119–134. Springer, 2011.

N. A. Danielsson. Total parser combinators. In *Proc. of ICFP 2010*, ICFP '10, pages 285–296. ACM, 2010.

N. A. Danielsson and The Agda Team. The Agda standard library, version 0.7, 2013.

S. Dolan. Fun with semirings: A funct. pearl on the abuse of linear algebra. In *Proc. of the 18th ACM SIGPLAN International Conf. on Funct. Prog.*, ICFP '13, pages 101–110. ACM, 2013.

D. Firsov and T. Uustalu. Certified parsing of regular languages. In *Certified Programs and Proofs*, pages 98–113. Springer, 2013.

D. Firsov and T. Uustalu. Certified CYK parsing of context-free languages. *J. Log. Algebr. Meth. Program.*, 83(5-6):459–468, 2014.

J. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *ESOP 2012*, pages 397–416, 2012.

M. Lange and H. Leiß. To CNF or not to CNF? an efficient yet presentable version of the CYK algorithm. *Informatica Didactica (8)(2008–2010)*, 2009.

L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. of the ACM (JACM)*, 49(1):1–15, 2002.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In *Mathematics of Program Construction*, volume 5133/2008 of *LNCS*, pages 268–283. Springer, 2008. doi: 10.1007/978-3-540-70594-9_15.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009. doi: 10.1017/S0956796809007345.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Tekniska Högskola, 2007.

A. Okhotin. Parsing by matrix multiplication generalized to boolean grammars. *Theor. Comp. Sci.*, 516(0):101 – 120, 2014.

T. Ridge. Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle. In *Soft. Language Engineering - 7th International Conf., SLE 2014, Västerås, Sweden, September 15-16, 2014. Proc.*, pages 261–281, 2014.

L. Valiant. General context-free recognition in less than cubic time. *J. of computer and system sciences*, 10(2):308–314, 1975.